ARMY RESEARCH LABORATORY

Context-Sensitive Compression of Database Updates for Wireless Transmission

by Frederick S. Brundick
and George W. Hartwig, Jr.

ARL-TR-1701 June 1998

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5067

ARL-TR-1701 **June 1998**

Context-Sensitive Compression of Database Updates for Wireless Transmission

Frederick S. Brundick, George W. Hartwig, Jr.
Information Science and Technology Directorate, ARL

Abstract

This report presents ways of encoding database to database updates (DDUs) to minimize communications bandwidth. The methods discussed are flexible and more easily extended than bit-oriented protocols (BOPs). The strengths and weaknesses of DDUs and BOPs are discussed. Typical DDUs are shown and the various data types sent in messages are explained. An analysis is performed to empirically determine the savings obtained by using the encoding methods. Finally, a decoder that generates structured query language (SQL) DDUs is presented as an example of the applicability of the algorithms to act as DDU translators.

Table of Contents

		Page
	List of Figures	v
	List of Tables	v
1.	Introduction	1
2.	Data Formats	2
2.1	Bit-Oriented Messages	2
2.2	Database Update Format	3
3.	Binary Encoding	5
3.1	Message Header	5
3.2	Binary Message Types	6
3.2.1	Binary Enumerated Fields	6
3.2.2	Usage Bit Vector	7
3.3	Field Types	7
3.3.1	Float	7
3.3.2	Integer	8
3.3.3	Factid	8
3.3.4	String	8
3.3.5	List	9
3.3.6	Binary Data	9
4.	Analysis	10
4.1	General Comments	10
4.2	Header	10
4.2.1	Binary Enumerated Fields	10
4.2.2	Usage Bit Vector	11
4.3	Values	11
4.3.1	Float	11
4.3.2	Integer	11
4.3.3	Factid	11
4.3.4	String	11
4.3.5	List	12
4.3.6	Binary Data	12
4.4	Empirical Results	13
4.4.1	ASCII Enumerated Fields	13
4.4.2	Binary Enumerated Fields	14
4.4.3	Usage Bit Vector	14
4.4.4	Message Sizes	15
5.	Further Enhancements	15
6.	Conclusions	16

7. References 19

Appendix: Combat Vehicle Command and Control (CVC2) Message Header . . 21

Distribution List 33

Report Documentation Page 37

List of Figures

Figure		Page
1.	Typical IDT DDU	1
2.	Enumerated ASCII Message	4
3.	Minimal Update Message	5
4.	Grid Fact Definition	5
5.	Binary Message Formats	6
6.	Simple Data Types	7
7.	Complex Data Types	8
8.	IDT DDU With a List Field	12
9.	Actual Messages	13
10.	IDT DDU Translated Into SQL	17

List of Tables

Table		Page
1.	Encoded Temperatures	2
2.	Uniform Temperatures	3
3.	Message Sizes and Ratios	14
A-1	CVC2 Message Header Field Types	23
A-2	CVC2 Message Header Fields	24

INTENTIONALLY LEFT BLANK.

1. Introduction

Information distribution technology (IDT) is a long-term research project of the Communications and Network Systems Division of the Information Science and Technology Directorate (IS&TD) of the U.S. Army Research Laboratory (ARL). The purpose of IDT research is to develop new concepts for the exchange of digital data over the low-bandwidth, combat net radio channels characteristically found on the tactical battlefield. These concepts make use of the computational power of the modern digital computer and the tenets of model-based battle command.

Model-based battle command is so named because each node on the battlefield maintains a database or model of the battle space and uses automated distribution rules to transmit information to specified destinations (Chamberlain 1995). Naturally, this leads to the information being distributed in the form of database-to-database updates (DDU). A typical IDT DDU is shown in figure 1. (Spaces and line breaks have been added to improve legibility.)

```
update @"0x803f5f3c/0x23" grid {
    east=407945; north=4368070;
    time=72; owner=@"0x803f5f3c/0x5";}
```

Figure 1. Typical IDT DDU.

One of the first things that becomes immediately obvious is the large size of the aforementioned exchange, especially when compared to a typical message from a bit-oriented protocol (BOP). The BOPs in use today are normally composed of messages that are often only 50 to a few hundred bits long; the sample DDU shown contains 89 bytes or 712 bits. DDUs can match that brevity but only when considered as a set of messages that might be sent over a period of time. To understand why, one must realize that BOPs obtain their terseness by limiting the area of interest, quantizing the data fields into short lists of possible values that are indexed by small integers, and using bit offsets within the message to indicate field names. This limits the information that can be sent and requires that the entire message be sent each time. The appendix shows the header for a combat vehicle command and control (CVC2) message (General Dynamics Land Systems 1991). This BOP was a precursor to inter-vehicular information system (IVIS) (U.S. Armor School 1995) and is used to show the flavor of bit-oriented messages.*

DDUs, on the other hand, use named fields that have numbers or strings as values. The primary advantages of this technique are that the data values do not need to be quantized to fit in the message format and only pertinent information needs to be sent. The latter fact alone may result in considerable savings in bandwidth.

This report examines various ways of compacting messages for efficient transmission. One extreme is the use of BOPs, where almost everything is sent as a coded value. Another

*A bit-oriented protocol defines a set of bit-oriented messages.

approach is to send DDUs, which are both terse and flexible but have a sizable amount of overhead. Two ways of reducing the size of DDUs are also discussed, and typical messages are examined.

2. Data Formats

2.1 Bit-Oriented Messages

The characteristics of bit-oriented messages that pose the most difficulty to tactical battle command are

- the quantization of data fields,
- the necessity of sending an entire message, and
- the inflexibility and difficulty involved in updating or changing the protocol.

To obtain the short list of possible meanings for a particular field, it is necessary to limit the precision. For example, suppose a system needs to store the current air temperature. It is necessary to represent values from about −50° to 140°, a range of 190°. At least 8 bits are needed to encode this range.* Alternatively, the temperature could be encoded with the scheme shown in table 1. Using these enumerated values, it is no longer possible to tell exactly what the temperature is. While the message contains an approximation of the value, its size has been reduced from 8 to 3 bits.

Table 1. Encoded Temperatures

Temp. (°F)		Description	Code
Min.	Max.		
120	140	dangerously hot	7
100	119	real hot	6
85	99	hot	5
55	84	nice	4
32	54	cool	3
0	31	cold	2
−20	−1	real cold	1
−50	−21	dangerously cold	0

The message fields in BOPs are named by their offset from the beginning of the message. This means that the entire message must be sent even if only a single field is being updated. Sometimes the number of fields in a message may be modified because of a value in an earlier

*$2^8 = 256$, which is slightly larger than 190.

field; however, generally this is not done. The space savings in enumerating all field values is partially offset by the requirement that all the fields be sent, regardless of the number that actually contain new values.

Another problem with BOPs is the fact that each system defines its own mappings. There are a lot of legacy systems in the field that cannot communicate with each other (Hartwig 1995). It is almost as if each was speaking a different language.

For example, suppose the system that defined the previously mentioned temperature codes needed to exchange information with a system that used the encoding scheme shown in table 2. The temperatures have been uniformly divided into 25° ranges. If an application was written to translate bit-oriented messages from one system to the other, how should it work? With the exception of the extreme temperatures, every temperature range in one system overlaps two or more ranges in the other system. Suppose the temperature was 42°. The first system would encode that as the value "3" because it is in the 32–54° range. What should the translator do with that value? Does it send it to the second system as code 3 or 4? There is no correct answer because the original value of 42 has been thrown away.

Table 2. Uniform Temperatures

Temp. (°F)		Code
Min.	Max.	
125	149	7
100	124	6
75	99	5
50	74	4
25	49	3
0	24	2
−25	−1	1
−50	−26	0

Even in the same system, there are communication problems because of extensions that have been made to various bit-oriented messages. One application needs to be able to refer to a certain type of platform, such as a rocket launcher. The second application, which is using the same bit-oriented protocol, receives the message but does not have that code value in its tables. Thus, it does not know what the message means.

2.2 Database Update Format

All messages in the workbench IDT system are sent as ASCII strings, surrounded by a wrapper. This approach makes it very easy to monitor the system, and the body of a message may be displayed or stored in a log file. However, such messages have two major disadvantages—they are not efficient, and binary data (such as images) may not be sent unless they are encoded in some way.

The smaller a message, the less bandwidth will be used. The IDT methodology of sending database updates only when absolutely necessary already makes good use of the limited bandwidth available to combat net radios. A typical DDU is shown in figure 1. This message is saying that the grid fact whose unique factid is 0x803f5f3c/0x23 is located at the point 407945, 4368070 at time 72. The grid is owned by fact 0x803f5f3c/0x5. This may seem cryptic, but it is a message that is being passed between two computer applications, not two humans. As presented to a human user, the message might appear as:

> The current location of Alpha Unit is the UTM coordinates 407945, 4368070 at the time 72 seconds into the mission.

The database update version is relatively terse and easily generated automatically by an application. It may just as easily be broken apart at the receiving end and the components stored in the database. However, it is keyword driven—the message contains pairs of field names and values—and the keywords take up a lot of space.

One solution is to enumerate all the fields and send those numbers instead of the field names. The obvious disadvantage is the fact that the receiver needs to know how the fields are numbered (e.g., is "north" field 2 or field 8 in a grid fact). All databases are given identical fact type definitions, so this problem goes away. If a database knows what a grid fact is, then it can enumerate the fields and be in agreement with everyone else. In a similar way, the various fact types must be assigned code numbers. Unlike the field types within a fact, there is no structure that lists all the fact types in the database. These values would be chosen *a priori* and distributed along with the fact type definitions.

A program was written in Perl (Wall, Christiansen, and Schwartz 1996) to test this idea. Figure 2 shows the same grid message (fact code 1) as described previously. Field 00 (east) has the value 407945, field 01 (north) equals 4368070, etc. In addition to enumerating the fields and replacing them with two ASCII digits, the program also compacts factid values. For example, @"0x803f5f3c/0x5" becomes 803f5f3c5. The database command `update` was coded by the single character "u".

```
u803f5f3c23 1{00407945;014368070;0272;15803f5f3c5}
```

Figure 2. Enumerated ASCII Message.

Suppose the unit has not moved, and it is time for it to send a message saying that it is still operational. The only field that has changed in its grid fact is the time, so an update with just one field may be sent. Figure 3 shows the DDU and its enumerated message.

The second form of compression is to encode the values in a more compact manner. For the program to properly decode a packed value, it must know what type of value it is. The type is easily obtained from the fact definition. The definition for a grid fact is shown in figure 4. Note that while the definition has 16 fields, only 4 of them were used in the example update message. Bandwidth is conserved by sending only the values that have changed, not the entire fact data structure.

```
update @"0x803f5f3c/0x23" grid {time=117;}

u803f5f3c23 1{02117}
```

Figure 3. Minimal Update Message.

```
define grid {
        int         east;
        int         north;
        int         time;
        int         soc_rad;
        int         deglat;
        int         minlat;
        float       seclat;
        string      hemilat;
        int         deglon;
        int         minlon;
        float       seclon;
        string      hemilon;
        int         terr1;
        int         terr2;
        int         usage;
        reference   owner;
};
```

Figure 4. Grid Fact Definition.

3. Binary Encoding

3.1 Message Header

Every message contains a database command, a factid, and its type. All messages currently in use are either updates or rupdates. An update supplies new values for an existing fact, while a rupdate creates a new fact with the initial values provided. The first picture in figure 5 shows the format of a binary message header.

The first bit is set if the message is a binary sequence rather than the old-style ASCII string. Since the high bit of all ASCII characters is zero, this approach is backward-compatible with the current message format. An ASCII string will have a zero for the first bit, which will tell the program to parse the message the old way.

The next bit is a mode bit to signify which of two binary coding schemes was used. They are discussed in detail in the next subsection. Bits 3 and 4 are used to hold the database command, allowing for four commands. Each additional bit doubles the number

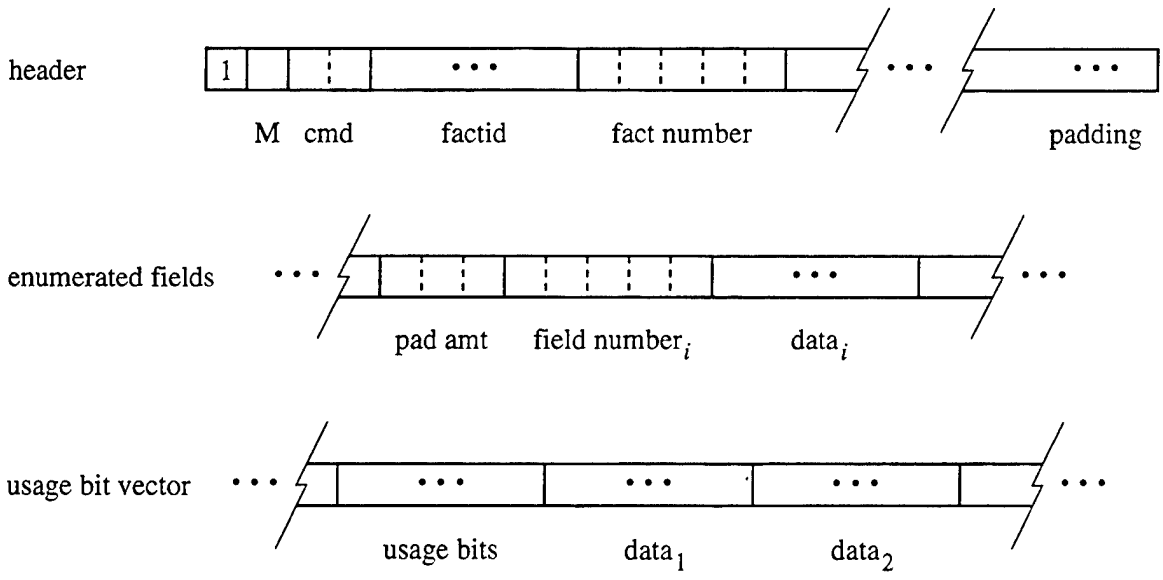

Figure 5. Binary Message Formats.

of commands, with negligible overhead. Since only two commands are used by existing messages, two bits were felt to be enough. Next in the header is the factid of the fact being updated. The format and number of bits required is described in section 3.3.3. The final piece of information is a block of 5 bits that holds the code. A 5-bit number may store the values 0–31, so this system could have up to 32 different fact types.

A message is made of bytes, while the encoded flags, counters, and values ignore byte boundaries. Therefore, the end of the message must be padded with zeroes to fill the last byte. The ramifications of this are discussed shortly.

3.2 Binary Message Types

As mentioned previously, there are two kinds of binary messages. Bit 2 of a binary message is set (one) if binary enumerated fields are used and is zero if a usage bit vector is supplied. The second picture in figure 5 shows the format for a binary enumerated field message, and the third picture is a usage bit vector message.

3.2.1 Binary Enumerated Fields

ASCII enumerated fields have been previously discussed and shown in figures 2 and 3. With binary enumeration, each encoded value is preceded by 5 bits that hold the field number. This allows each fact type to have a maximum of 32 fields.

Nothing in an enumerated message gives the total number of bits or the number of fields in the message. The wrapper used by the fact exchange protocol (FEP)* does include the

*The FEP is the protocol developed for the workbench IDT system.

length of the message, so the decoder knows how many bytes are in the message. A block of 3 bits is inserted between the fact number and the first field/value pair. It contains the number of bits that should be ignored in the last byte of the message.

3.2.2 Usage Bit Vector

A usage bit vector is a block of bits, one for each field of the fact definition. The left-most bit corresponds to field zero, the next to field one, and so on, until all the fields have a placeholder. If the bit is set, it means there is a value in the message for that field. The usage vector appears after the fact type number.

The decoder knows how many values are in the message; it is the same as the number of bits set in the usage vector. Since the number of bits used by the encoded values may be determined from the fact types (as with enumerated fields), the decoder does not need to be explicitly told the number of padding bits. It will stop decoding the message when it has processed all the fields.

3.3 Field Types

Field types are divided into two categories: simple and complex. Simple fields may vary slightly in size but only between fixed bounds. Complex fields encode their size and will appear in multiple blocks if they exceed the "maximum" length. This is explained in detail in section 3.3.4. The three simple and two complex formats are shown in figures 6 and 7, respectively.

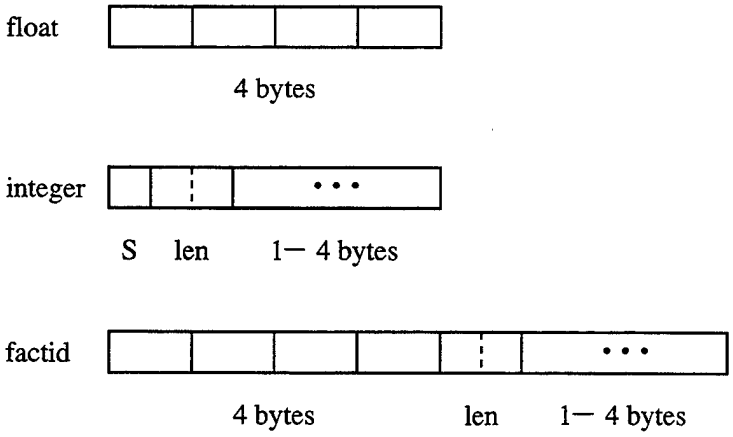

Figure 6. Simple Data Types.

3.3.1 Float

Floating-point numbers are the easiest values to encode. They always consist of 4 bytes (32 bits) and no special processing is required. They are denoted by **float** in a fact definition.

3.3.2 Integer

Integers (**int**) are also stored in the computer in 4 bytes. However, while the float format always uses 4 bytes, an integer is padded to the left with zeroes. The number may be compressed by sending as few bytes as needed. A single byte is used for the values 0–255, two bytes for 256–65535, etc. Therefore, 2 bits are prepended to each integer to tell how many additional bytes (beyond the minimum of one) are used to encode the number. In other words, the bits `00` mean a 1-byte integer follows, `01` means 2 bytes, `10` means 3 bytes, and `11` is used for 4 bytes.

Negative integers are a special case. Because of sign extension, all 4 bytes are always used internally. Rather than send 32 bits for a small value like −17, a signed format is used. A sign bit in front of the length bits is set if the number is negative, and the absolute value of the integer is encoded in the value bytes. The value −42 would require 32 bits if the internal format was sent but only 11 bits (`1 00 00101010`) with the described method.

3.3.3 Factid

All factids (type **reference**) in the entire system must be unique. This was achieved by splitting a factid into two parts—a host and a counter—each half containing 4 bytes or 32 bits. The hostid in the workbench system is the computer's Internet address, which is a unique 32-bit value. Each computer keeps its own counter and increments whenever it creates a new fact. The combination of hostid and counter is therefore a unique value.

All 64 bits could blindly be sent, but the counter portion is like an integer, and thus the same compression may be used.* A factid is encoded as 4 bytes, 2 length bits, and 1–4 additional bytes, for a total of 42 to 66 bits.

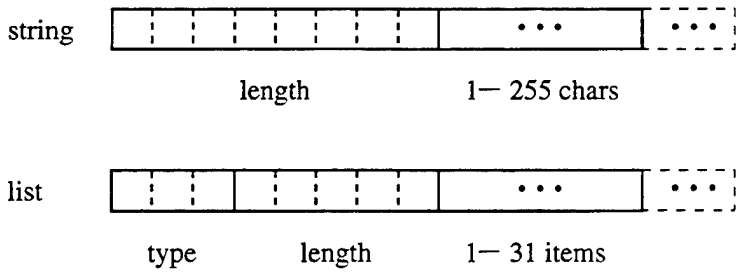

Figure 7. Complex Data Types.

3.3.4 String

A **string** is a sequence of 7-bit characters. There is no arbitrary limit to how long a string may be, except a null or empty string is never used in a message.†

*Of course, the sign bit is not needed.

†Rather than set a field to a null string, the application would delete the field.

As shown in figure 7, the encoded string is preceded by its length, which is stored in 8 bits. This allows a maximum string length of 255 characters, which is more than adequate for the messages being sent by current applications. However, there may be times when a larger string must be transmitted. In that case, the length bits will have the value zero. The first 255 characters of the string will be encoded, followed by another substring consisting of 8 length bits and some number of characters. The pattern will repeat until the entire string has been encoded.

The string itself will be packed when it is encoded. ASCII characters are 7-bit values, yet are stored internally in 8-bit bytes. Space may be saved in the binary message by throwing away the high bit (which is always zero) and encoding the remaining 7 bits. For example, if a string has a length of 9 characters, it requires 9 bytes or 72 bits internally (plus a terminating byte of zeroes, which may be ignored). The same string encoded in a message would have an 8-bit length prefix (containing the value 9) and 9×7 or 63 bits, for a total of 71 bits in the message.

3.3.5 List

Strictly speaking, a list is not a type but a container of other types. It is used in a fact definition when it is not known in advance how many fields may be needed in a fact, similar to the way it is impossible to predict how long a string will be. For example, a unit fact has exactly one superior, but may have any number of subordinate units.

The database would be more robust if there were subtypes of lists, such as list_int and list_reference. Instead, it is the responsibility of the application that accesses the list field to know what type of data it contains. While it is possible for a list to contain values of more than one type at the same time, the binary encoder assumes all lists are homogeneous.

The encoder does the same thing the current list processor does. It examines each value in the list and determines its type. With only four different types to choose from, this is a simple task. The format of a list consists of 3 bits that denote the type of the list; like the fact codes, the numbers are defined manually. The rest of the list format is identical to the string format, except only 5 bits are used for the list length. If the list contains more than 31 items, a zero length and the first 31 items are encoded, followed by a second sublist and group of values, etc.

If it is decided that mixed-type lists are going to be employed, then each value in the list will be prefixed by its type. This is a lot of overhead. An alternative would be to have a flag bit that is set if the list is uniform, in which case the type code would appear only once. If the binary message encoder/decoder is implemented in the FEP, lists will probably be redesigned both in the messages and in the database itself.

3.3.6 Binary Data

The database cannot directly store arbitrary binary data such as images. They must be converted to ASCII data and stored in strings in a process similar to the way a string is

decoded from a binary message. Binary data could be stored in the database as a sequence of bytes, plus an integer stating the number of bytes. A binary field in a message would be identical to a string, except more than 8 bits would be used for the length (binary data tends to be quite large) and there would be no packing of the data.

4. Analysis

4.1 General Comments

It is impossible to state what compression ratios the two binary schemes described will achieve. However, some general comments may be made before some empirical results are discussed. These comments apply to the ASCII messages currently being used, not the refined ASCII enumerated field format presented in section 2.2.

4.2 Header

A message always starts with a database command, a factid, and the name of a fact type. The command string is either `update` or `rupdate`, which is 6 or 7 characters (bytes) long, respectively. A typical factid is of the form `@"0xHHHHHHHH/0xHH"`, although the counter portion could be 4–8 hexadecimal digits instead of 2. There are 8 hex digits for the hostid, 2 or 4 for the counter, and 8 more characters (`@"0x.../0x"`) to tell the database and applications that this is a factid. The name of the fact is like any other programming variable. It should be long enough to be understandable, yet short enough to be reasonable. The fact types in the current system range from 3 to 10 characters in length, with 7 or 8 being a typical value. The ASCII header of a message thus contains from 27 to 41 characters, with a typical value of about 32 characters or 256 bits.

In contrast to this, a binary header starts with 2 mode bits and 2 command bits. The factid will be either 42, 50, 58, or 66 bits. The fact type code is always 5 bits. Padding must be added to the end of the message and ranges from 0 to 7 bits. A binary header therefore contains 51, 59, 67, or 75 bits, plus padding, or 7 to 10 bytes. The typical value is about 55 bits, or 7 bytes. In the header alone, the compression ratio may exceed 4:1.

4.2.1 Binary Enumerated Fields

An enumerated field message starts with 3 bits to encode the amount of padding at the end of the message. The length of the field names in the current database varies from 1 to 12 characters, with an average length of about 6 characters. The field name is separated from the field value by an equals sign, and the pair is terminated with a semicolon, adding two more bytes to the length of each field/value pair. Each field is identified by 5 bits instead of the 64 bits on average required by the ASCII message. The average savings is therefore $64n$ vs. $3 + 5n$, or about 10:1.

4.2.2 Usage Bit Vector

The bit vector format does away with the three padding length bits. It also eliminates the repeating 5-bit prefix to each value, replacing all of the prefixes with a single vector. The current fact types contain from 3 to 22 fields, meaning each bit vector contains 3–22 bits. If an average fact contains 10 fields and a message updates 5 fields, then the savings is 64×5 vs. 10 or 30:1.

4.3 Values

It appears as though converting the ASCII messages to binary will save a tremendous amount of bandwidth. The savings so far have been obtained by replacing ASCII strings (fact and field names) with either very small numbers or a single large bit vector. As this subsection explains, encoding values into a binary message is not nearly as efficient.

4.3.1 Float

A floating-point number is always encoded as 4 bytes. Any number whose ASCII representation is more than 4 characters will be compressed (e.g., `3.14159`, `-18.62`). For a compression ratio of only 2:1, the number must have 8 characters.

4.3.2 Integer

The size of an encoded integer may be 11, 19, 27, or 35 bits. Unlike a float, the value of an integer is just as important as the number of ASCII characters needed to represent it. The 3-character number `255` is encoded in 11 bits (2.2:1), while `256`, which also has 3 characters, requires 19 bits (1.3:1). Generally speaking, the "longer" the number, the better the compression ratio. The integer `-12345678` needs 27 bits instead of 72 or 2.7:1.

4.3.3 Factid

A typical factid, such as `@"0x803f5f3c/0x23"`, is 18 characters long (144 bits) and may be encoded in 42 bits. This is a compression ratio of 3.4:1. The worst case is a counter greater than 16,777,215, which drops the ratio to 2.9:1.

4.3.4 String

The simple packing method used to encode strings results in an 8:7 or 1.14:1 compression ratio for the characters making up the string. An additional byte is required to signify the length of the string; however, it replaces the double quotes that surround strings in the current message format. A string made of a single character (e.g., `"S"`) is 24 bits long and

is encoded in 15 bits; a 255-character string encodes as 1793 bits; and a 256-character string requires 1809 bits. The compression ratios are 1.6:1, 1.15:1, and 1.14:1, respectively. The longer the string, the closer it approaches the limit of 8:7.

4.3.5 List

It is difficult to generate numbers for list compression because it depends on what type of values the list contains. A search of the current fact type definitions shows that nearly all the lists contain factids. This is to be expected because a list frequently contains subordinate components that are other facts in the database. Figure 8 contains a message with a list field. The fact type line has a field that contains the list of line segments that make up the line.

```
update @"0x803f5f3c/0x20" line {stime=0; etime=524;
    type=1; usage=4; name="amber"; user=@"0x803f5f3c/0x5";
    line_seg=[@"0x803f5f3c/0x14" @"0x803f5f3c/0x15"
    @"0x803f5f3c/0x16" @"0x803f5f3c/0x17" @"0x803f5f3c/0x18"
    @"0x803f5f3c/0x19" @"0x803f5f3c/0x1a" @"0x803f5f3c/0x1b"
    @"0x803f5f3c/0x1c" @"0x803f5f3c/0x1d" @"0x803f5 f3c/0x1e"];}
```

Figure 8. IDT DDU With a List Field.

The list is surrounded by brackets (to tell the receiving database that a list is being sent), and the items are separated by spaces. The overhead for a list is thus $n + 1$ bytes or $8 \times (n + 1)$ bits. In the binary message, the overhead is 8 bits for a list of 1 to 31 items, 13 bits for 32 to 63 items, and so on. The compression ratio for the list overhead ranges from 2:1 for a list of only one item to 32:1 for 31 items, 20:1 for 32 items, and 39:1 for 63 items. However, like the total message overhead, the list overhead is insignificant compared to the size of the list values. The compression ratio of the example is 1680 bits vs. 470 bits, or 3.6:1.

4.3.6 Binary Data

Binary data would be encoded in a message as is with an integer holding its length preceding it. There would be no savings unless some kind of binary compression scheme was used. Currently, binary data must be encoded as ASCII data using a 6:8 expansion ratio. Packing the resultant ASCII string into a message would reduce the expansion to 6:7.

4.4 Empirical Results

The compression algorithms were implemented as Perl programs and applied to various actual messages, both to determine the space savings and to show the ideas work.* Some messages were from a battlefield simulation, while others were constructed directly from facts stored in the database. The messages are presented in figure 9, and the message sizes and ratios are in table 3.

```
update @"0x803f5f3c/0x22" sync {command=1; time=871580009;
    index=0; rate=1;}
update @"0x803f5f3c/0x21" dist_vars {thresh=2; host=@"0x803f5f3c/0x5";
    cmd=@"0x803f5f3c/0x5";}
update @"0x803f5f3c/0x5" unit_type {plan_loc=@"0x803f5f3c/0x23";
    act_loc=@"0x803f5f3c/0x24"; name="Alpha Unit";
    cur_rt=@"0x803f5f3c/0x20"; plans=[@"0x803f5f3c/0x20"];}
update @"0x803f5f3c/0x20" line {stime=0; etime=524; type=1;
    usage=4; name="amber"; user=@"0x803f5f3c/0x5";
    line_seg=[@"0x803f5f3c/0x14" @"0x803f5f3c/0x15" @"0x803f5f3c/0x16"
    @"0x803f5f3c/0x17" @"0x803f5f3c/0x18" @"0x803f5f3c/0x19"
    @"0x803f5f3c/0x1a" @"0x803f5f3c/0x1b" @"0x803f5f3c/0x1c"
    @"0x803f5f3c/0x1d" @"0x803f5f3c/0x1e" @"0x803f5f3c/0x1f"];}
update @"0x803f5f3c/0x14" line_seg {sgrid=@"0x803f5f3c/0x7";
    egrid=@"0x803f5f3c/0x8"; radius=100;}
update @"0x803f5f3c/0x23" grid {east=407945; north=4368070; time=72;
    soc_rad=100; usage=-3; owner=@"0x803f5f3c/0x5";}
rupdate @"0x803f5f43/0x5" sensing {desc="HOKM"; wd=2; tv=1; att=211;
    zhour=47600; sym="T1xxxMmHTRW.H211"; speed=170; east=52000;
    north=7000; num=1; window=["R1"]; dir=2862; len=200; mode=19;}
rupdate @"0x803f5f48/0x1a1" target {num1=36; active=1;
    tgtnum="09040.1"; rds1=@"0xc0051725/0xb6";
    tgt=@"0x803f5f48/0x179"; unit=@"0xc0051725/0x846";}
```

Figure 9. Actual Messages.

4.4.1 ASCII Enumerated Fields

The savings in the enumerated ASCII message are fairly minor, as expected. Integers and floats remain the same, and strings lose only two characters (the double quotes). Lists remain unchanged because the receiving database needs to be told what type of values the

*All messages were successfully decoded into their original forms.

Table 3. Message Sizes and Ratios

ASCII	ASCII Enum		Enumerated		Bit Vector	
Length	Bytes	Ratio	Bytes	Ratio	Bytes	Ratio
70	38	1.8	18	3.9	16	4.4
89	42	2.1	21	4.2	20	4.5
160	89	1.8	42	3.8	42	3.8
338	284	1.2	93	3.6	89	3.8
93	44	2.1	21	4.4	19	4.9
109	61	1.8	27	4.0	25	4.4
175	114	1.2	58	3.0	52	3.4
141	76	1.9	40	3.5	37	3.8

list contains. The type could have been coded in a single digit, allowing the values to be encoded where possible. However, the list would still require separators and delimiters. If this is done with the message containing the list of factids, the compression ratio climbs from 1.2 to 1.8, which matches the other messages.

The real savings are with factids, which drop by almost a factor of 2, and in the message overhead. The database command becomes a single character; the fact type is represented by a one or two digit number; and all field names, regardless of their lengths, are replaced by two digits.

On the other hand, the compression is not that good. The average ratio is under 2:1, and binary data may not be sent. Additional refinements, such as performing Huffman coding on the encoded ASCII message, would further reduce the message size.

4.4.2 Binary Enumerated Fields

Encoding the message in binary drops the final message size by another factor of 2, or an average compression ratio of almost 4:1. This method has all the advantages of the ASCII enumerated fields (with better ratios), plus it compresses all values as explained in section 4.3.

4.4.3 Usage Bit Vector

Replacing the 5 bits encoding each field name with a single bit vector reduces each message by 0–6 bytes. The best savings are in messages that contains several fields; the least are with messages that have few fields for facts that contain many fields. The number of bits used with enumeration is $3 + 5n$ vs. a fixed size for the bit vector. If n is small and the bit vector is long, very little (if anything) is gained.

4.4.4 Message Sizes

What is really important is not the compression ratio, per se, but the size of the message that is transmitted. A better compression algorithm results in a smaller message, as does more refinement in the way a message is constructed. A multipage text message describing an action on the battlefield may be reduced, using traditional text compression algorithms, to half or even a third of its original size, but it is still a large message that requires manual interpretation by the person who receives it.

Database updates are succinct. The sample messages are all 70–150 bytes long (with one major exception) and may be compressed into binary messages of 16–60 bytes. The exception is a message with a large list of values, and, even so, it is only 90 bytes long. These sizes are comparable to BOPs.

5. Further Enhancements

The data type with the worst compression ratio is the string. Using a standard text compression algorithm like Huffman or arithmetic coding would reduce the number of bits needed to send the string in a message. As with all optimization techniques, the data must be analyzed to make sure it is worth the effort. If most strings that are sent are fairly short, such as unit names, then the additional savings could be insignificant. An abundance of large text messages could mean that more fact types or fields are needed. Also, compressing and expanding the strings may be very costly in time and computational power.

The database command is transmitted as a code value. Both the sender and receiver must have a table to encode and decode the command. The formats discussed here allow 2 bits to store the command, which limits the system to only four commands. If new commands are added, the entire format must be changed.

Likewise, entire concepts may be enumerated and sent as messages, which is the basis for bit-oriented protocols. The more this is done, the smaller the message may be made, at least in theory. In practice, what happens is very specific messages are created for various applications. These messages are so tightly coupled to the applications that they are incompatible with other systems that have their own sets of messages and codes.

As mentioned previously, BOPs require that all the fields are sent in a given message, even if some of the values have not changed. The message does not shrink if fewer fields are needed. The result is like a usage bit vector message without the bit vector and a value for every field in the fact type.

Perhaps it would be possible, in certain circumstances, to leave off the hostid part of a factid. The values in a fact may all refer to the same hostid, as is the case with most of the examples in figure 9. A flag bit could be reserved in the message header; if the bit is set, then all factids have the same host as the factid in the header. Such a scheme may cross the line between efficiency and confusion. Is it really necessary to compress messages any further than the 4:1 ratio?

Including binary data in a message may not be as simple as it appears. Strings are relatively short and are easily included in a message. Binary data, especially images, tend to be very large, even with current image-compression techniques. The protocol being used to send the messages allows several messages to be packed and sent out together. A large binary image would have to be split apart and sent as several packets. These messages may arrive at their destination in the wrong order and must be reassembled correctly. This implies more overhead for binary fields and possibly changes to the fact exchange protocol.

If a bit is assigned the wrong value in an ASCII message, the error might be easily detected. For example, changing a single bit from 0 to 1 turns the digit "6" into the letter "v". With a binary message, an incorrect or dropped bit may drastically affect the meaning of the entire message. Forward error correction codes must be included in the protocol to ensure that messages are not corrupted by the transmission process. This adds still more overhead to the message, a problem that is common to all protocols that perform binary compression, such as BOPs.

6. Conclusions

Communications bandwidth on the battlefield is severely limited, requiring digital messages to be as small as possible. Bit-oriented protocols reduce the message size by enumerating fields into code values where possible. However, BOPs require that the entire message always be sent, even if some of the fields are empty. BOPs are also difficult to change and result in a variety of legacy systems that may not communicate with each other.

DDUs are a concise way of distributing information around the battlefield in a form that is readily used by computers. Since the updates are keyword-driven, the field names take up space, which offsets the savings made by not sending all the fields. The binary encoding schemes proposed here reduce the size of a DDU by a factor of 3 or 4. With a typical size of only 16–60 bytes (plus overhead required by the FEP), binary database updates may be transmitted with a minimum of bandwidth. It should be noted, however, that the most efficient compression algorithms will not allow DDUs to work well over combat net radio networks if complete audit trails are required. Any interruption of service will result in a backlog of old updates.

Unlike hardwired formats like bit-oriented messages, most of the encoding is completely dynamic, with the encoder and decoder knowing very little about the actual database. Any message system that uses field/value pairs may be encoded with these algorithms. In fact, a Perl program was written to prove this point. It reads the binary messages and converts them to SQL (Structured Query Language), the standard database language (Date 1986). Figure 10 shows an IDT DDU and the corresponding SQL as generated by the decoder.

Messages being exchanged on the battlefield must be both terse and understandable. The most efficient compression scheme in the world is useless if only certain applications know how to decode the messages. The method used must be portable, expandable, and adaptable. On today's battlefield, interoperability is very important. Stand-alone systems

```
update @"0x803f5f3c/0x23" grid {east=407945;
    north=4368070; time=72; soc_rad=100; usage=-3;
    owner=@"0x803f5f3c/0x5";}

update grid
    set east=407945,
        north=4368070,
        time=72,
        soc_rad=100,
        usage=-3,
        owner=@"0x803f5f3c/0x5"
    where factid = @"0x803f5f3c/0x23";
```

Figure 10. IDT DDU Translated Into SQL.

are no longer desirable, not just within one service, but between services as well. Information about units detected by ground forces must be sent to support aircraft, a nontrivial task with current legacy systems.

Applications should not rely solely on improved ways to send messages, whether by sophisticated compression schemes or improved communications hardware. They should examine the messages they are sending and consider alternatives. Is a verbose text message required, or can a handful of database updates convey the same information? Should updates be sent out periodically, or will the use of models reduce bandwidth by limiting the amount of message traffic? With these thoughts in mind, a binary encoded database update is a very efficient way of conveying information on the battlefield.

INTENTIONALLY LEFT BLANK.

7. References

Chamberlain, Samuel C. "Model-Based Battle Command: A Paradigm Whose Time Has Come." *First International Symposium on Command and Control Research and Technology*, pp. 31–38, Washington, DC, June 1995.

Date, C. J. *An Introduction to Database Systems, Volume I.* Reading, MA: Addison-Wesley Publishing Company, 1986.

General Dynamics Land Systems Division. "Combat Vehicle Command and Control Program Interface Control Document." ICD-SY10001A, Warren, MI, February 1991..

Hartwig, George W., Jr. "Automated Translation of Bit-Oriented Messages (BOMs) into Data Kernel Representations (DKR)." Technical Report ARL-TR-728, U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, 1995.

U.S. Armor School. "The Digitized Battalion Task Force." Fort Knox Supplemental Material No. 71-2-1, Fort Knox, KY, December 1995.

Wall, Larry, Tom Christiansen, and Randal L. Schwartz. *Programming Perl.* Sebastopol, CA: O'Reilly & Associates, 1996.

INTENTIONALLY LEFT BLANK.

Appendix:

Combat Vehicle Command and Control (CVC2) Message Header

INTENTIONALLY LEFT BLANK.

This appendix contains the specification for a combat vehicle command and control (CVC2) message header. CVC2 was a bit-oriented message protocol precursor of the inter-vehicular information system (IVIS) and, although newer protocols exist, this example is sufficient to show how such protocols work. The field types are shown in table A-1. In table A-2, the start bit is the offset from the beginning of the message to where the field begins. Field width is the number of bits devoted to this field.

Table A-1. CVC2 Message Header Field Types

enum	A list of permitted values are defined, and the field value is used as an index into that list.
m_enum	Several enumerated lists exist and the one used depends on the value of the previous field. The current field value is then used as an index into the appropriate list.
na	This field is not currently used and is being saved for future expansion.
num	This field is to be interpreted as a integer number.
char	This field is to be interpreted as an ASCII character.

Table A-2. CVC2 Message Header Fields

Start Bit	Field Type	Field Width	Field Name	Possible Values
0	enum	2	"MSG_TYP"	"text"
				"overlay"
				"spare"
				"spare"
2	m_enum	6	"MES_DESC"	"Reserved"
				"MOPP Status Alert"
				"Air Alert"
				"REDCON Alert"
				"NBC Alert"
				"Warning Order"
				"Frago"
				"Call For Fire"
				"Call For CAS"
				"Contact Report"
				"Engagement Update"
				"Spot Report"
				"Situation Report"
				"Bridge Report"
				"Minefield Laying Report"
				"Obstacle Report"
				"Route Report"
				"Personnel Status"
				"Ammo Status Report/Request"
				"POL Status Report/Request"
				"Vehicle Status"
				"NBC 1 Observers Report"
				"NBC 4 Contamination Report"
				"NBC 5 Contamination Area Report"
				"Shell, Bomb, Mortar Report"
				"Strikewarn"
				"Position Update"
				"WILCO"
				"Request For Reports"
				END
				"Reserved"
				"Own Current Operations Overlay"
				"Own Future Operations Overlay"
				"Higher Current Operations Overlay"
				"Higher Future Operations Overlay"
				"Enemy Overlay"
				"Enemy Overlay Update"
				"Obstacle Overlay"
				"Obstacle Overlay Update"
				"Fire Support Overlay"
				"Fire Support Overlay Update"
				"Fire Plan"
				"Sector Identification"
				END

Table A-2. CVC2 Message Header Fields (continued)

Start Bit	Field Type	Field Width	Field Name	Possible Values
8	na	6	"spare"	
13	enum	1	"WILCO"	"WILCO not required"
				"WILCO required"
				END
14	enum	2	"POS_REP"	"two words"
				"three words"
				END
16	num	16	"MSG_WORD_CNT"	/* valid 1-7650 */
32	enum	4	"SND UNIT"	"Reserved"
				"Company A"
				"Company B"
				"Company C"
				"Company D"
				"Company E"
				"Company F"
				"Company G"
				"Company H"
				"Company I"
				"Company J"
				"Company K"
				"Company L"
				"Battalion"
				"Adjacent Battalion"
				"Brigade"
				END
36	m_enum	2	"SND ELEMENT"	/* Company */
				"Headquarters"
				"Headquarters"
				"Front"
				"Headquarters"
				END
				/* Battalion */
				"1st Platoon"
				"Slice"
				"Rear"
				"Slice"
				END
				/* Adjacent Battalion */
				"2nd Platoon"
				"Admin/Log"
				"Left"
				"Admin/Log"
				END

Table A-2. CVC2 Message Header Fields (continued)

Start Bit	Field Type	Field Width	Field Name	Possible Values
36	(cont)			/* Brigade */ "3rd Platoon" "spare" "Right" "spare" END
38	m_enum	3	"SND INDIV"	/* Company HQ */ "Reserved" "Co 1st Sgt" "FIST_V" "Engineering Squad" "ADA Squad" "Co Cmdr" "Co Exec" END /* 1/2/3/PLATOON */ "Reserved" "Platoon Ldr" "Wingman A" "Wingman B" "Platoon Sgt" "spare" "spare" "spare" END /* BN & BDE HQ */ "Reserved" "S1" "S2" "S3" "S4" "Bn Sgt Maj" "Bn Cmdr" "Bn Exec" END /* BN & BDE SLICE */ "Reserved" "Attack Helicopter" "Aviation Plt" "Air Defense Arty" "Heavy Mortar" "Scout" "ENGINEER" "spare" END

Table A-2. CVC2 Message Header Fields (continued)

Start Bit	Field Type	Field Width	Field Name	Possible Values
38	(cont)			/* BN ADMIN/LOG */
				"Reserved"
				"Maintenance"
				"Medic"
				"Support"
				"spare"
				"spare"
				"spare"
				"spare"
				END
				/* BDE ADMIN/LOG */
				"Reserved"
				"Maintenance"
				"Medical"
				"Supply Support"
				"Transportation"
				"Commo Officer"
				"spare"
				"spare"
				END
41	na	3	"spare"	
44	enum	4	"SND ADD ID"	"US"
				"GERMANY"
				"UNITED KINGDOM"
				"FRANCE"
				"NETHERLANDS"
				"DENMARK"
				"CANADA"
				"BELGIUM"
				END
48	enum	4	"DST UNIT"	"Reserved"
				"Company A"
				"Company B"
				"Company C"
				"Company D"
				"Company E"
				"Company F"
				"Company G"
				"Company H"
				"Company I"
				"Company J"
				"Company K"
				"Company L"
				"Battalion"
				"Adjacent Battalion"
				"Brigade"
				END

Table A-2. CVC2 Message Header Fields (continued)

Start Bit	Field Type	Field Width	Field Name	Possible Values
52	m_enum	2	"DST ELEMENT"	/* Company */ "Headquarters" "Headquarters" "Front" "Headquarters" END /* Battalion */ "1st Platoon" "Slice" "Rear" "Slice" END /* Adjacent Battalion */ "2nd Platoon" "Admin/Log" "Left" "Admin/Log" END /* Brigade */ "3rd Platoon" "spare" "Right" "spare" END
54	m_enum	3	"DST INDIV"	/* Company HQ */ "Reserved" "Co 1st Sgt" "FIST_V" "Engineering Squad" "ADA Squad" "Co Cmdr" "Co Exec" END /* 1/2/3/PLATOON */ "Reserved" "Platoon Ldr" "Wingman A" "Wingman B" "Platoon Sgt" "spare" "spare" "spare" END

Table A-2. CVC2 Message Header Fields (continued)

Start Bit	Field Type	Field Width	Field Name	Possible Values
54	(cont)			/* BN & BDE HQ */
				"Reserved"
				"S1"
				"S2"
				"S3"
				"S4"
				"Bn Sgt Maj"
				"Bn Cmdr"
				"Bn Exec"
				END
				/* BN & BDE SLICE */
				"Reserved"
				"Attack Helicopter"
				"Aviation Plt"
				"Air Defense Arty"
				"Heavy Mortar"
				"Scout"
				"ENGINEER"
				"spare"
				END
				/* BN ADMIN/LOG */
				"Reserved"
				"Maintenance"
				"Medic"
				"Support"
				"spare"
				"spare"
				"spare"
				"spare"
				END
				/* BDE ADMIN/LOG */
				"Reserved"
				"Maintenance"
				"Medical"
				"Supply Support"
				"Transportation"
				"Commo Officer"
				"spare"
				"spare"
				END
57	na	3	"spare"	

Table A-2. CVC2 Message Header Fields (continued)

Start Bit	Field Type	Field Width	Field Name	Possible Values
60	enum	4	"SND ADD ID"	"US"
				"GERMANY"
				"UNITED KINGDOM"
				"FRANCE"
				"NETHERLANDS"
				"DENMARK"
				"CANADA"
				"BELGIUM"
				END
64	na	2	"spare"	
66	num	6	"SND_GZN"	
72	char	8	"SND_GZL"	
80	char	8	"100KM SQ_COL"	
88	char	8	"100KM SQ_ROW"	
96	num	16	"EASTING"	
112	num	16	"NORTHING"	
128	num	5	"DTG DATE"	
133	num	5	"DTG HOUR"	
138	num	6	"DTG MINUTE"	
144	num	6	"DTG SECOND"	
150	enum	5	"DTG MONTH"	"spare"
				"JANUARY"
				"FEBRUARY"
				"MARCH"
				"APRIL"
				"MAY"
				"JUNE"
				"JULY"
				"AUGUST"
				"SEPTEMBER"
				"OCTOBER"
				"NOVEMBER"
				"DECEMBER"
				"spare"
				END
155	enum	5	"DTG TIME ZONE"	"spare"
				"ALPHA"
				"BRAVO"
				"CHARLIE"
				"DELTA"
				"ECHO"
				"FOXTROT"
				"GOLF"
				"HOTEL"
				"JULIET"
				"KILO"

Table A-2. CVC2 Message Header Fields (continued)

Start Bit	Field Type	Field Width	Field Name	Possible Values
155	(cont)			"LIMA" "MIKE" "NOVEMBER" "PAPA" "QUEBEC" "ROMEO" "SIERRA" "TANGO" "UNIFORM" "VICTOR" "WHISKEY" "X-RAY" "YANKEE" "ZULU" "LOCAL" "spare" END

INTENTIONALLY LEFT BLANK.

NO. OF COPIES ORGANIZATION

2 DEFENSE TECHNICAL INFORMATION CENTER
DTIC DDA
8725 JOHN J KINGMAN RD
STE 0944
FT BELVOIR VA 22060-6218

1 HQDA
DAMO FDQ
DENNIS SCHMIDT
400 ARMY PENTAGON
WASHINGTON DC 20310-0460

1 DPTY ASSIST SCY FOR R&T
SARD TT F MILTON
RM 3EA79 THE PENTAGON
WASHINGTON DC 20310-0103

1 OSD
OUSD(A&T)/ODDDR&E(R)
J LUPO
THE PENTAGON
WASHINGTON DC 20301-7100

1 CECOM
SP & TRRSTRL COMMCTN DIV
AMSEL RD ST MC M
H SOICHER
FT MONMOUTH NJ 07703-5203

1 PRIN DPTY FOR TCHNLGY HQ
US ARMY MATCOM
AMCDCG T
M FISETTE
5001 EISENHOWER AVE
ALEXANDRIA VA 22333-0001

1 DPTY CG FOR RDE HQ
US ARMY MATCOM
AMCRD
BG BEAUCHAMP
5001 EISENHOWER AVE
ALEXANDRIA VA 22333-0001

1 INST FOR ADVNCD TCHNLGY
THE UNIV OF TEXAS AT AUSTIN
PO BOX 202797
AUSTIN TX 78720-2797

NO. OF COPIES ORGANIZATION

1 GPS JOINT PROG OFC DIR
COL J CLAY
2435 VELA WAY STE 1613
LOS ANGELES AFB CA 90245-5500

3 DARPA
L STOTTS
J PENNELLA
B KASPAR
3701 N FAIRFAX DR
ARLINGTON VA 22203-1714

1 US MILITARY ACADEMY
MATH SCI CTR OF EXCELLENCE
DEPT OF MATHEMATICAL SCI
MDN A MAJ DON ENGEN
THAYER HALL
WEST POINT NY 10996-1786

1 DIRECTOR
US ARMY RESEARCH LAB
AMSRL CS AL TP
2800 POWDER MILL RD
ADELPHI MD 20783-1145

1 DIRECTOR
US ARMY RESEARCH LAB
AMSRL CS AL TA
2800 POWDER MILL RD
ADELPHI MD 20783-1145

3 DIRECTOR
US ARMY RESEARCH LAB
AMSRL CI LL
2800 POWDER MILL RD
ADELPHI MD 20783-1145

ABERDEEN PROVING GROUND

4 DIR USARL
AMSRL CI LP (305)

NO. OF COPIES	ORGANIZATION
1	HQDA USAAIC ARMY 107 PENTAGON WASHINGTON DC 20310-0107
1	DISC4 ARMY 107 PENTAGON DAIS ADO WASHINGTON DC 2010-0107
1	CDR ARMY DIGITIZATION OFFICE HQDACS ADO 1745 JEFFERSON DAVIS HWY CRYSTAL SQUARE 4 STE 403 ARLINGTON VA 22202
2	CDR USAARMC ATSB CG ATZD CDS MAJ POWERS FORT KNOX KY 40121
1	CDR USAARMC ATZK MW FORT KNOX KY 40121
1	CDR USAIC ATSB CG FORT BENNING GA 31905
1	CDR TOPOGRAPHIC ENGRG CTR CETEC TD AG 7701 TELEGRAPH RD ALEXANDRIA VA 22315-3864
1	CDR HQ TRADOC ATCS X BLDG 133 FORT MONROE VA 23651-5000

NO. OF COPIES	ORGANIZATION
1	CDR HQ TRADOC ATCD ZA FORT MONROE VA 23651-5000
1	CDR USAICS ATZS CD FORT HUACHUCA AZ 85613-6000
2	CDR USASIGCEN ATZH CDC ATZH BLT FORT GORDON GA 30905
1	CMDT USAFASCH ATSF CBL FORT SILL OK 73503-5600
1	PEO CCS SFAE CC SEO FORT MONMOUTH NJ 07703-5207
1	PM AFAS SFAE CC INT TSE 1616 ANDERSON RD MCLEAN VA 22102-1616
1	PM FATDS SAFAE CC FS TMD FORT MONMOUTH NJ 07703-5207
1	PM OPTADS SFAE CC MVR TM FORT MONMOUTH NJ 07703-5207
1	DIR AMSEL RD CS BC CC 2 SALTON CECOM FORT MONMOUTH NJ 07703-5207

NO. OF COPIES	ORGANIZATION
3	CMDR TPIO ABCS COL R HARTEL LTC R JONES BCBL L LTC D MACGREGOR FORT LEAVENWORTH KS 66027-2300
1	CMDR US ARMY TOPOGRAPHIC ENGNRG CTR CETEC TD AG S MICHAEL 7701 TELEGRAPH RD ALEXANDRIA VA 22315-3864
1	DANIEL H WAGNER ASSOC R OVERTON STE 500 2 EATON ST HAMPTON VA 23669
1	CUBIC DEFENSE SYSTEMS INC C SOLINSKY PO BOX 85587 SAN DIEGO CA 92186-5587
1	NOVA RESEARCH CORP 203 MIDDLESEX TURNPIKE STE 6 BURLINGTON MA 01803

NO. OF COPIES	ORGANIZATION
	ABERDEEN PROVING GROUND
11	DIR USARL AMSRL IS J D GANTT AMSRL IS CB L TOKARCIK AMSRL IS CI B D BROOME T P HANRATTY AMSRL IS T J GOWENS AMSRL IS TP F S BRUNDICK H CATON S C CHAMBERLAIN G W HARTWIG M C LOPEZ C SARAFIDIS

INTENTIONALLY LEFT BLANK.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1998	Final, 1 Jul - 1 Sep 97

4. TITLE AND SUBTITLE
Context-Sensitive Compression of Database Updates for Wireless Transmission

5. FUNDING NUMBERS
622618.H80
8TE510
BFTA0

6. AUTHOR(S)
Frederick S. Brundick and George W. Hartwig, Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: AMSRL-IS-TP
Aberdeen Proving Ground, MD 21005-5067

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-1701

9. SPONSORING/MONITORING AGENCY NAMES(S) AND ADDRESS(ES)

10.SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report presents ways of encoding database to database updates (DDUs) to minimize communications bandwidth. The methods discussed are flexible and more easily extended than bit-oriented protocols (BOPs). The strengths and weaknesses of DDUs and BOPs are discussed. Typical DDUs are shown and the various data types sent in messages are explained. An analysis is performed to empirically determine the savings obtained by using the encoding methods. Finally, a decoder that generates structured query language (SQL) DDUs is presented as an example of the applicability of the algorithms to act as DDU translators.

14. SUBJECT TERMS
data compression, computers, communications, modeling, and databases

15. NUMBER OF PAGES
37

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

INTENTIONALLY LEFT BLANK.

USER EVALUATION SHEET/CHANGE OF ADDRESS

This Laboratory undertakes a continuing effort to improve the quality of the reports it publishes. Your comments/answers to the items/questions below will aid us in our efforts.

1. ARL Report Number/Author ARL-TR-1701 (Brundick) Date of Report June 1998

2. Date Report Received _____

3. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which the report will be used.) _____

4. Specifically, how is the report being used? (Information source, design data, procedure, source of ideas, etc.) _____

5. Has the information in this report led to any quantitative savings as far as man-hours or dollars saved, operating costs avoided, or efficiencies achieved, etc? If so, please elaborate. _____

6. General Comments. What do you think should be changed to improve future reports? (Indicate changes to organization, technical content, format, etc.) _____

CURRENT ADDRESS

_____
Organization

_____
Name E-mail Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

7. If indicating a Change of Address or Address Correction, please provide the Current or Correct address above and the Old or Incorrect address below.

OLD ADDRESS

_____
Organization

_____
Name

_____
Street or P.O. Box No.

_____
City, State, Zip Code

(Remove this sheet, fold as indicated, tape closed, and mail.)

(DO NOT STAPLE)